

Programming Support for Speculative Execution with Software Transactional Memory



Min Feng

NEC Laboratories America

Rajiv Gupta

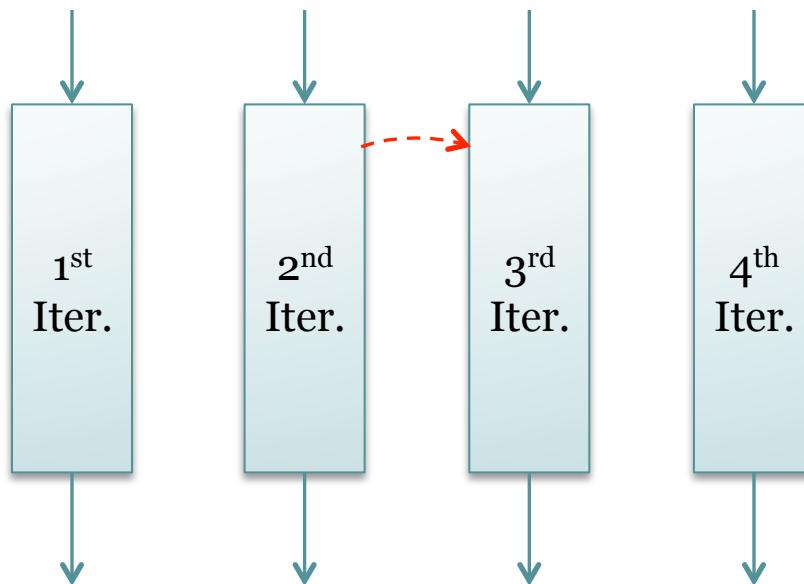
University of California, Riverside

Iulian Neamtiu

University of California, Riverside

Loop Parallelization

Thread 1 Thread 2 Thread 3 Thread 4



**Speculative parallelization
with
Software Transaction Memory**

Practical Issues

- Larger transactions => excessive instrumentation
 - Hundreds of potential shared accesses
 - Many functions called inside transactions
- Library functions
 - Source code not available
 - Sometimes irreversible

Transaction Representation

- Conceptual form

```
// insert a node into a linked list
atomic {
    node = head;
    while (node->next != NULL &&
        node->val != newnode->val)
        node = node->next;
    insert(newnode, node);
}
```

```
void insert(Node *newnode, Node *node) {
    newnode->next = node->next;
    node->next = newnode;
}
```

Real case ???

STM Libraries

```
// insert a node into a linked list
{
    _begin_tx(txDesc);           // <-- transaction boundary
    node = _tx_read(txDesc, &head);          // <-- shared read
    while (_tx_read(txDesc, &node->next) != NULL &&
        _tx_read(txDesc, &node->val) != _tx_read(txDesc, &newnode->val))
        _tx_write(txDesc, &node, _tx_read(txDesc, &node->next));    // <-- shared write
    insert(newnode, node, txDesc);
    _end_tx(txDesc);           // <-- transaction boundary
}

void insert(Node *newnode, Node *node, TxDescriptor *txDesc) {      // <-- pass descriptor
    _tx_write(txDesc, &newnode->next, _tx_read(txDesc, &node->next));
    _tx_write(txDesc, &node->next, newnode);
}
```

For each STAMP Benchmark, on average 177 STM constructs are added and 2750 lines of code need to be examined.

Intel STM Compiler

```
// insert a node into a linked list
__tm_atomic {
    node = head;
    while (node->next != NULL &&
        node->val != newnode->val)
        node = node->next;
    insert(newnode, node);
}

__attribute__((tm_callable))      // <-- annotate function header
void insert(Node *newnode, Node *node) {
    newnode->next = node->next;
    node->next = newnode;
}
```

For each STAMP benchmark, there are on average 58 functions called inside transactions.

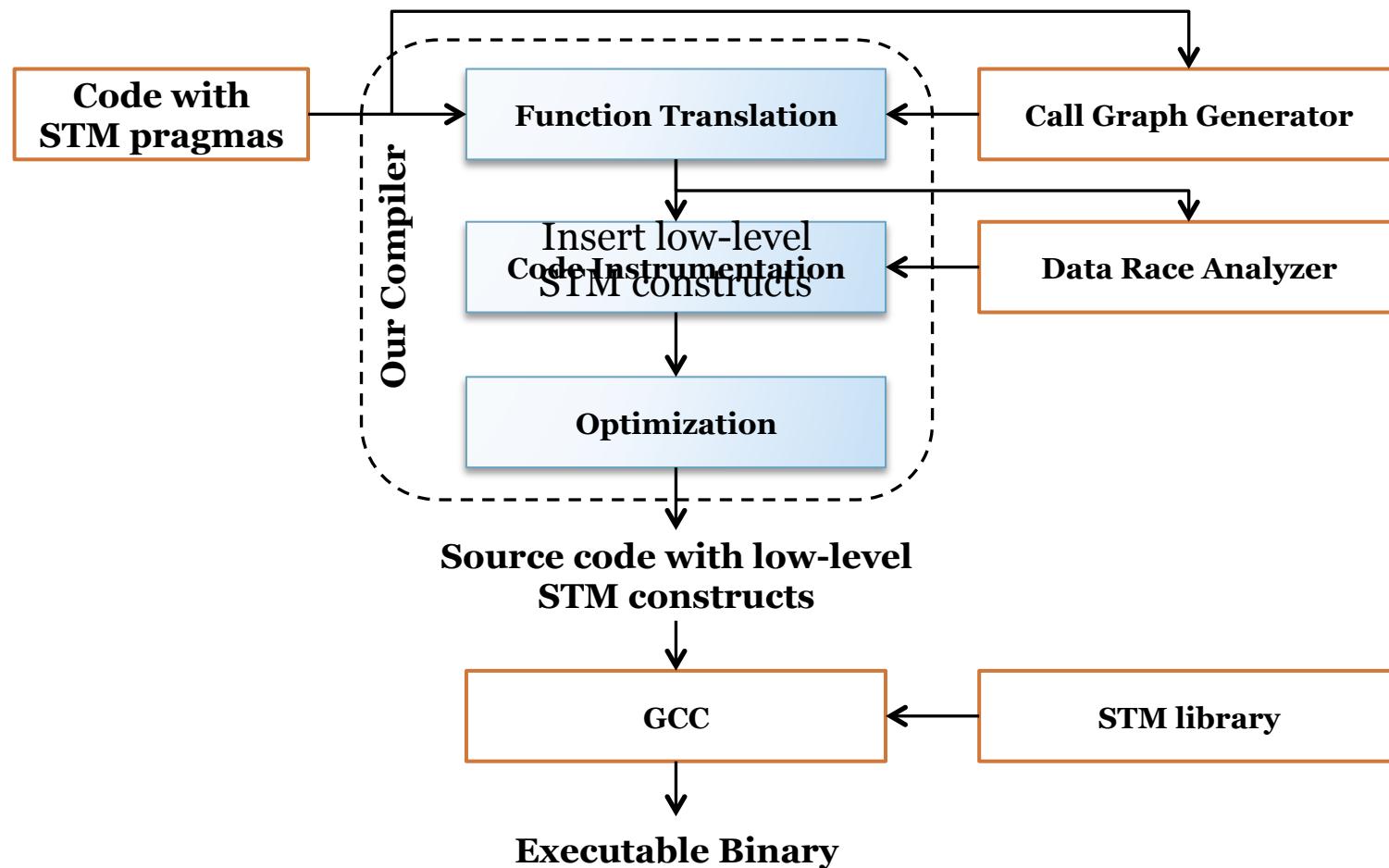
Our Constructs

```
#pragma tm transaction
{
    node = head;
    while (node->next != NULL &&
        node->val != newnode->val)
        node = node->next;
    insert(newnode, node);
}

void insert(Node *newnode, Node *node) {
    newnode->next = node->next;
    node->next = newnode;
}
```

- Our constructs do not require:**
- 1. Inserting shared read/write barriers.**
 - 2. Annotating functions called inside transactions.**

Code Generation for Copying/Commit



Function Translation

- Classification
 - Atomic functions
 - Non-atomic functions
 - Double-duty functions
 - Dynamically-called functions

```
int original_func()  
{  
    if ( inside_transaction == true )  
        return atomic_func();  
    // original statements start here  
}
```

Code Instrumentation

1. Find shared variables
 - Use static data race analysis
2. Normalize operators
 - $b=++a \Rightarrow b=(a=a+1)$
 - $b=a++ \Rightarrow b=(t=a, a=a+1, t)$
3. Insert Barriers
 - $a=a+1 \Rightarrow _tx_write(txDesc, &a,$
 $_tx_read(txDesc, &a) +1)$

Optimizations

- Eliminate redundant read/write barriers

Original code	Intermediate code	Generated code
b=a+1; b=a*b;	_tx_write(txDesc, &b, _tx_read(txDesc, &a)+1); _tx_write(txDesc, &b, _tx_read(txDesc, &a)* _tx_read(txDesc, &b));	t_a=_tx_read(txDesc, &a); t_b=t_a+1; t_b=t_a*t_b; _tx_write(txDesc, &b, t_b);

Safety

- Aliasing of shared variable
 - **Problem:** Multiple temporary variables may be created for a variable pointed to by multiple pointers, thus causing data inconsistency
 - **Solution:** Kill temporary variables when a new pointer is dereferenced
- Exception handling
 - **Problem:** An exception may happen in speculatively parallel regions due to misspeculation
 - **Solution:** Atomicity checks and commits need to be performed before throwing the exception

Library Functions

- Two types of library functions cannot be transact-ified
 - Precompiled library functions
 - Source code not available
 - System calls and I/O operations
 - Cannot be rolled back

Precompiled Functions

- Code example

```
char src[LEN], dst[LEN];
#pragma tm transaction
{
    #pragma tm precompiled \
        read(dst, (*src, size), src, size) \
        write((*dst, size))
    memcpy(dst, src, size);
}
```

- Translated code

```
// create local copies
int l_size = stmReadInt(tx, &size);
void *l_src = (void*)malloc(size);
void *l_dst = (void*)malloc(size);
stmReadBytes(tx, (void*)l_src,
            (void*)src, l_size);
// execute the function call
memcpy(l_dst, l_src, l_size);
// update shared variables
stmWriteBytes(tx, (void*)dst,
              (void*)l_dst, l_size);
free(l_src); free(l_dst);
```

Irreversible Functions

- Code example

```
char ch;  
#pragma tm transaction  
{  
    #pragma tm suspend read(ch)  
    putchar(ch);  
}
```

- Translated code

```
stmBegin(tx);  
// store arguments and functions  
args.sharedpush(tx, &c, sizeof(char));  
funcs.push( F_PUTCHAR );  
stmEnd(tx);  
...  
// call function after the transaction  
// succeeds  
void resume_suspended_funcs() {  
    while ( !funcs.empty() )  
        switch ( funcs.pop() ) {  
            case F_PUTCHAR:  
                putchar( *((char*)args.pop()) );  
                break;  
            ...  
        } }
```

Evaluation

- Our source-to-source translator was built on top of ROSE.
- We use the TL2 STM library.
- Platform: Dell Poweredge T605 (Intel Xeon 8-core 2.0Ghz) running CentOS v5.5
- Benchmarks: STAMP benchmarks + two real applications – Velvet and Incremental Tree Inducer.

Programming Effort

Benchmark	STM Lib	Intel STM	Ours
Bayes	176	75	15
Labyrinth	98	31	3
Genome	122	31	5
Intruder	265	95	3
Kmeans	17	3	3
Ssca2	55	10	10
Vacation	359	151	3
Yada	324	115	6

97% fewer programming constructs inserted compared to low-level STM API

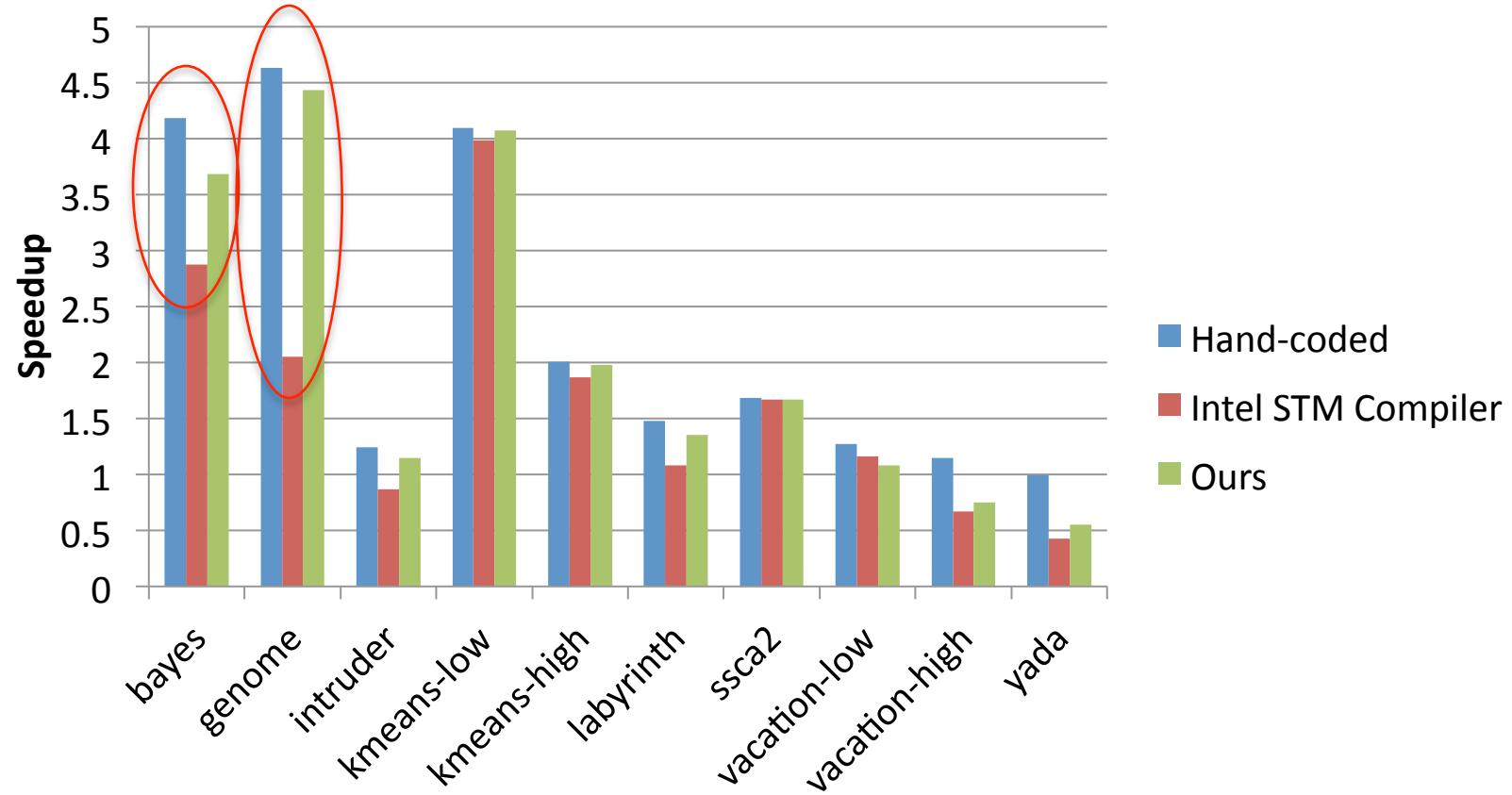
91% fewer programming constructs inserted compared to the Intel STM compiler

Impact of Data Race Analysis

Benchmark	Baseline	DRA
Bayes	172	87
Labyrinth	107	67
Genome	40	40
Intruder	183	152
Kmeans	7	7
Ssca2	24	24
Vacation	196	180
Yada	424	304

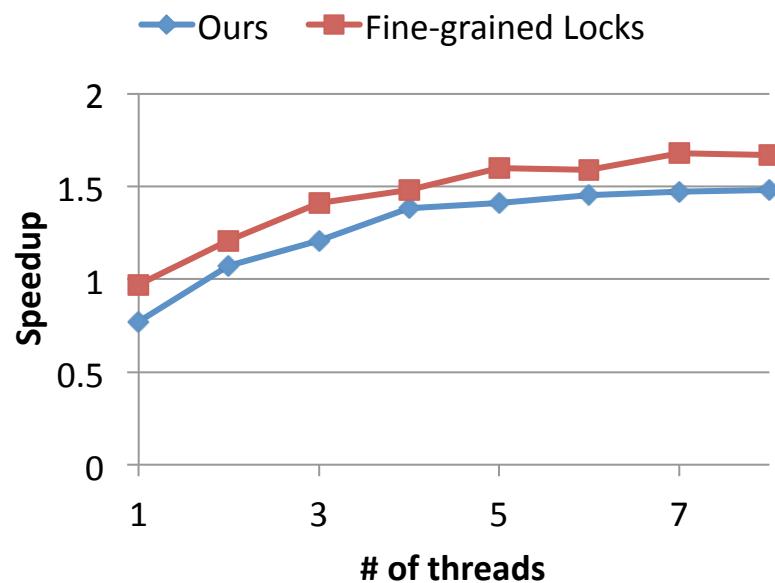
DRA reduces the number of barriers by **25.3%** on average.

Performance

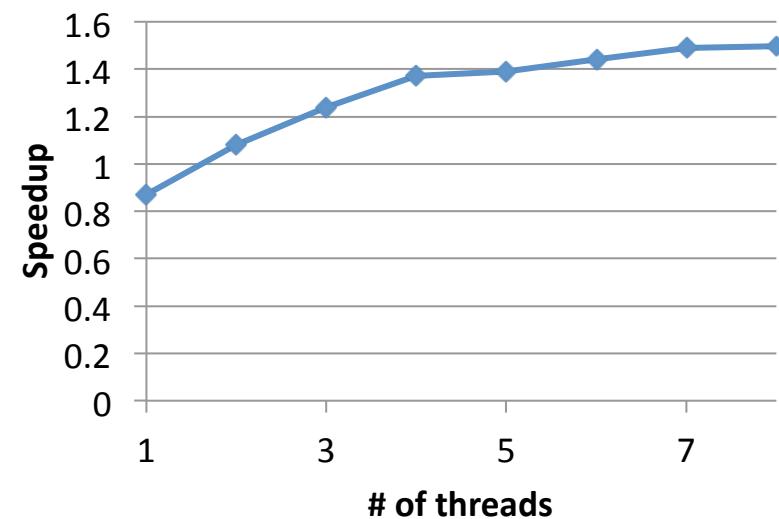


We achieve **1.6x** speedup on average

Performance on Real Applications



(a) Velvet



(b) Incremental Tree Inducer

Conclusion

- Speculative parallelization with existing STM libraries and compilers is not an easy task.
 - Excessive instrumentation
 - Library functions
- Our compiler addresses these issues
 - Only require marking transaction boundaries
 - Support precompiled and irreversible library functions
 - Optimize the code with data race analysis and redundant read/write barriers elimination